UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingeniería

Desarrollo de un motor de juegos online con arquitectura cliente servidor basado en envío de evento

Proyecto Integrador

Fausto Iván Zamora Arias

Ingeniería en Sistemas

Trabajo de titulación presentado como requisito
para la obtención del título de
Ingeniero en Sistemas

Quito, 12 de mayo de 2017

UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

COLEGIO DE CIENCIAS E INGENIERÍA

HOJA DE CALIFICACIÓN
DE TRABAJO DE TITULACIÓN

Desarrollo de un motor de juegos online con arquitectura cliente servidor basado en envío de eventos

Fausto Iván Zamora Arias

Calificación:	95/100
Nombre del profesor, Título académico	Aldo Cassola, PhD

Firma del profesor

_____

Quito, 12 de mayo de 2017

Derechos de Autor

Por medio del presente documento certifico que he leído todas las Políticas y Manuales de la Universidad San Francisco de Quito USFQ, incluyendo la Política de Propiedad Intelectual USFQ, y estoy de acuerdo con su contenido, por lo que los derechos de propiedad intelectual del presente trabajo quedan sujetos a lo dispuesto en esas Políticas.

Asimismo, autorizo a la USFQ para que realice la digitalización y publicación de este trabajo en el repositorio virtual, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior.

Firma del estudiante: _____

Nombres y apellidos: Fausto Iván Zamora Arias

Código: 00107240

Cédula de Identidad: 1720895455

Lugar y fecha: Quito, mayo de 2017

RESUMEN

El presente trabajo describe el desarrollo de un motor de juegos basado en arquitectura de cliente-servidor y su análisis de rendimiento de memoria y uso del CPU. Un motor de juegos puede definirse como el sistema encargado de manejar los componentes para que un juego funcione. De acuerdo a esto se han hecho distintos módulos para integrarlos en un sólo sistema, estos módulos han sido: el modelo de objetos, sistema de eventos, sistema de red, y sistema de actualización del hilo principal. Cabe mencionar que el enfoque del trabajo es que los sistemas y módulos mencionados se integren de una manera remota en red y es por esto que el sistema de red es parte principal del sistema. Para lograr dicho enfoque se ha dividido al motor de juego en procesos de cliente y procesos de servidor. Los procesos de cliente avisan de nuevos eventos, generados por el usuario, al servidor y este se ha de encargar de ejecutarlos y actualizarlos para los demás clientes. Además, se han hecho pruebas para medir el rendimiento del sistema tanto para clientes como en servidor de memoria y de uso del CPU, las cuales han mostrado un uso casi constante en el uso de estas. Sin embargo, la experiencia de usuario en LAN debería pulirse en un futuro pues es uno de los temas destacables para satisfacción de un juego que se haga con el motor y es que los usuarios puedan experimentar en varios medios de red. Es así como se propone un sistema de motor de juegos capaz de trabajar en red con arquitectura cliente-servidor.

Palabras clave: Motor de Juegos, video juegos, red, cliente, servidor, eventos, modelo de objeto de juego.

ABSTRACT

This paper describes the development of a Game Engine based on a Client-Server architecture and its analysis of memory performance and CPU usage. A game engine can be defined as the system in charge of handling the components for a game to work. According to this, different modules have been made to integrate them into a single system, these modules have been: the object model, event system, network system, and main thread update system. It is worth mentioning that the approach for this work is that the system modules become integrated remotely in network, that is why the network system is a main part of the engine. In order to achieve the approach the game engine has been divided into client processes and server processes. The client processes warns of new events, generated by the user, to the server and it must manage the execution of them and update them for the other clients. In addition, tests have been made to measure the performance of the system for both clients and server for memory and CPU usage, which have shown almost constant use of these. However, the user experience in LAN should be polished in the future as it is one of the highlights for satisfaction of a game that is done with the engine and because users may be able to experiment, in the future, in various network media. In this way, It is proposed a gaming engine system capable of working in a network with client-server architecture.

Key words: Game Engine, videogames, networking, client, server, events, game object model.

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

INTRODUCCIÓN

A lo largo de la historia de los videojuegos se han hecho y programado técnicas para su mejor implementación en el hardware, así mismo las computadoras han evolucionado y dichas técnicas han evolucionado. Un ejemplo de esta evolución es el uso de paradigmas computacionales como la programación orientada a objetos que permite la separación de propiedades y métodos de un elemento de un programa, de otros y otorga independencia para posibles cambios en el comportamiento de este elemento. Una de estas técnicas fue hecha en la década de 1990 en el popular juego DOOM, juego de disparos en primera persona, la cual consistía en dar una separación clara entre los componentes principales del sistema (eventos, organización de componentes, sistemas de inteligencia artificial, manejo de gráficas y simulación de física) de los componentes propios del juego(reglas de juego, modelos de personajes, diseño de niveles , entre otros). Al resultado de manejo del software se lo llamó motor de juegos. Formalmente un motor de juegos se define como un sistema extensible que es base para varios juegos de video diferentes sin que se hagan modificaciones importantes al sistema (Gregory, 2014). Sin embargo, a criterio de este trabajo, se debe tener clara las características que se desean transmitir en el sistema, es decir que es lo que va a hacer. Para tener claro que desea tener un motor de juegos se debe saber a qué tipos de juegos va a ser orientado y para esto se tiene que tomar en cuenta los principios de las mecánicas de juego que se utilizan para distintos géneros de juego (Roberts, 2015). Es así como este trabajo trata del desarrollo de un motor de juegos basándose en principios de arquitectura de motores, diseño de juegos por computadora y diseño de sistemas.

La arquitectura de un motor de juegos sigue las mismas pautas del diseño de sistemas clásico, en referencia a aquel diseño de sistemas que no está enfocado a juegos, pero con la diferencia de enfocarse exclusivamente a la sincronización de múltiples sistemas en tiempo real. En general un sistema de juegos debe manejar varios subsistemas para que se sincronizan unos con otros en un lazo principal, esto puede lograrse por medio del uso de arquitecturas multiprocesador (Gregory, 2014). Según Sommerville las arquitecturas multiprocesador son un modelo simple de sistemas distribuidos en el que el sistema tiene un número de procesos diferentes que podrían ejecutarse en separado. El número de procesos que un juego puede ejecutar depende de los sistemas que este requiera, para esto se tiene tres modelos para el diseño: Un hilo por subsistema, por *Jobs* (trabajos), y programación asincrónica. La técnica de diseño de un hilo por subsistema es un enfoque de multitarea para asignar subsistemas particulares del motor para que se ejecuten en hilos separados, un hilo maestro se encarga de controlar y sincronizar las operaciones de los hilos secundarios para que estos compartan información que mantenga la lógica del juego, pero al hacer esto cada hilo depende del trabajo de otro con una tarea específica para poder llamar al siguiente y puede representar restricciones en cómo le hilo sistema va a ser utilizado. En el caso del diseño por Jobs es dividir el trabajo en pequeños hilos independientes llamados Jobs o trabajos, este enfoque puede ser resumido como piscina de hilos (*thread pool*). Por último, el diseño enfocado en programación asíncrona se refiere al hecho de que cuando un proceso requiere de resultados de una operación perteneciente a otro, este no va a estar habilitado inmediatamente después de la petición de este (Gregory, 2014). Dados estos enfoques y definiciones anteriormente expuestas se puede generalizar que un motor de juegos debe ser concebido como un sistema distribuido con multitareas y por lo tanto debería ser catalogado como un sistema de tiempo real (Sommerville, 2000).

Por otra parte, los motores de juego manejan programas que representan sistemas suaves que no son críticos para la ejecución de transacciones o procesos, en otras palabras manejan videojuegos cuya ejecución no depende de datos críticos como por ejemplo las transacciones de dinero que suelen hacer lo bancos, pues el objetivo de un videojuego es hacerse ver como experiencia interactiva que sea divertida. Según Gregory un motor de juegos es un sistema que permite crear diferentes tipos de juegos sin embargo no son completamente perfectos para cualquier tipo de juego, por lo cual existen motores especializados para juegos de peleas, juegos de disparos en primera persona, de rol, entre otros. Estos géneros ayudan a definir las mecánicas y sistemas que se han de construir en el motor pues se puede considerar que los requerimientos de los géneros de juegos son estándares que deben cumplir sus respectivos motores. Los géneros mejor definidos en la industria de videojuegos son los *First-Person Shooters*(FPS): Las características de los juegos FPS deben ser enfocadas en ambientes de tres dimensiones con animaciones de alta fidelidad para personajes e inteligencia artificial para enemigos; Plataformas o juegos de tercera persona: para el género de plataformas se debe contar con objetos que representen plataformas o artefactos para interactuar con el jugador en ambientes estilo rompecabezas, y algoritmos de colisión de cámara para ambientes en tres dimensiones; juegos de pelea: debe contar con un conjunto rico en animaciones de pelea, detecciones de colisión muy precisas y un sistema de detección de combinaciones de botones; juegos de carreras: uso de algoritmos de trucos visuales para simular el movimiento de objetos, uso de estructuras de datos para la representación del camino y algoritmos avanzados de control de cámara ; juegos en línea de multijugadores masivos: Necesidad de servidores conectados que mantengan la lógica del juego para varios jugadores, manejo de transacciones de dinero de los jugadores, y soporte para muchísimos jugadores conectados

al mismo tiempo ;y de contenido generado por el jugador: son juegos que requieren que el jugador cree, publique y comparta lo que ha hecho en el mundo de juego, un ejemplo de este género es *Minecraft* (Gregory, 2014). Así es que un motor de juego no puede abarcar todos los tipos de géneros existentes, pues cada uno tiene necesidades diferentes que muchas veces no son desarrolladas completamente pues ese no sería el objetivo del motor en particular.

Estos principios expuestos son base para la creación de juegos y de motores, y en este trabajo se han cubierto aspectos como el uso de las mecánicas para juegos de plataforma y multiprocesamiento. Las mecánicas de un juego de plataformas pueden varias dependiendo de lo que se quiera lograr en el mundo, pero el hecho de que un motor de juegos sea específico de ese género requiere que esté preparado para lo que va a querer el diseñador del juego. Teniendo en cuenta esto el trabajo anteriormente expuesto, se ha hecho el suficiente esfuerzo programático para complacer esas necesidades, pero para un ambiente en dos dimensiones. Estos esfuerzos incluyen el desarrollo de una arquitectura distribuida con un enfoque cliente servidor, pero con un servidor que aplica principios de trabajo con multiprocesadores. Por otra parte, el hecho de que esté especializado en plataformas no significa que no se pueden hacer otros tipos de juegos, un ejemplo de esto es *Unity* que es un motor comercial que ha permitido la realización de varios géneros con su misma arquitectura principal (Unity, 2017), esto se lo dejaría como objetivo secundario pero muy posible debido al principio de extensibilidad del sistema.

DESARROLLO DEL TEMA

Como se ha descrito anteriormente, un motor de juegos usa varios subsistemas de manera que necesita de una arquitectura que permita el manejo y la sincronización de múltiples procesos. Es por esto que se ha decidido desarrollar el proyecto como un sistema distribuido, específicamente con enfoque de arquitectura cliente-servidor. Sin embargo, se ha tomado en consideración que al estar ejecutado varios sistemas en paralelo, se debería tener un control de acuerdo a como estos sistemas vayan actuando, es por esto que se ha decidido sumar una arquitectura de manejo de eventos compartida entre el cliente y el servidor, permitiendo así el uso de prioridades. Por otra parte, para la representación de mecánicas y objetos en el sistema se ha optado por un modelo de diseño que permita la extensibilidad de implementar nuevas cualidades a los personajes y objetos del juego. Es así como el desarrollo del motor de juegos se dividió en tres partes: Diseño de objetos con mecánicas de juego, sistema de eventos y desarrollo de arquitectura de red para múltiples jugadores. Cabe mencionar que el diseño del proyecto está basado en los deberes del curso de fundamentos de motores de juego (*Game Engine Foundations*) del periodo de otoño del 2015 por el profesor David Roberts en la Universidad estatal de Carolina del Norte.

Objetos y mecánicas de juego

Una de las partes importantes de un sistema de juegos es cómo se manejan y representan las estructuras de objetos de juego. Esto se logra abstrayendo conceptos de programación que puedan dar maneras de consultar las relaciones que los objetos tienen con otros tipos o acceder a la información del tipo de objeto en tiempo de ejecución o para propósitos de depuración (DeLoura, 2001). Para lograr una abstracción satisfactoria se ha considerado los elementos básicos que tienen los juegos en general, pero en concentración al género de plataformas, pues el motor pretende ser para uso de desarrollo de este género.

Según Schell existen cuatro elementos básicos que un juego posee, estos son las mecánicas, la historia, la estética y la tecnología. Este trabajo se enfoca en la parte de la tecnología pues trata del motor y del sistema que permite la interacción. Sin embargo, para el diseño de los objetos de juego que se representarán en el motor se necesita entender los otros elementos, en especial la parte de las mecánicas pues estas son las que dan las reglas y el proceso de juego, además describen lo que le jugador puede o no hacer (Schell, 2008). Así mismo se consideró que la estética de los juegos puede ser implementada en el sistema por defecto, es por esto que se ha implementado además una manera de representar los objetos a manera de cuadrados, tal como se puede ver en el juego "*Thomas was alone*" que los personajes son completamente rectangulares y los ambientes también (Bithell, 2012). En el caso de la historia, se ha considerado que el motor de juegos no debería manejar la historia pues es un elemento que el director o diseñador de un juego debería tomar más en cuenta que lo que haría el diseñador del motor, sin embargo, eso no quita la posibilidad de que se pueda usar el presente motor para narrar una historia interactiva.

Con los elementos anteriores se ha dispuesto a hacer un diseño de acuerdo al modelo de objeto de juego por componentes. Un componente es una parte que es modular, instalable y reemplazable del sistema que encapsula una implementación y expone una serie de interfaces (Roger S & Bruce R, 2015), este modelo de objetos tiene su fundamento en la programación orientada a objetos en el cual cada característica que va a tener un elemento se lo llama componente (Gregory, 2014). Se puede entender el modelo de componentes para un juego con el siguiente ejemplo: Se dice que se tiene un personaje que puede caminar, disparar y saltar, para el modelo se puede decir que el objeto de juego es el personaje y que sus componentes son caminar, disparar y saltar (Roberts, 2015). Para implementar dicho modelo en el motor se ha creado una clase llamada "`GameObj`" el cual tiene referencia a componentes de uso posibles que se han diseñado en base a una interfaz llamada "`Component`". Se ha decidido implementar cuatro características para los objetos de juego, las cuales se ha convenido que son las importantes para el tipo de juego que el motor está enfocado, estas son que un objeto sea dibujable, movible, chocable y chocable como una zona muerta, esta última para añadir elementos de rompecabezas en los ambientes. Cabe mencionar que estos componentes son opcionales pues dependiendo de cómo se desee hacer un juego con el motor. Por otro lado, según los principios de arquitectura de motores, para todo juego existe un lazo principal que se encarga de actualizar los datos de los objetos de juego de acuerdo a cambios en el ambiente o ejecución de eventos (Gregory, 2014). Es por esto que el diseño de los objetos y componentes debe tomar en cuenta este principio. Así que tanto la clase de objetos de juego como la interfaz de componentes tienen métodos de actualización("`Update`") . Cada componente tiene una acción específica. EL componente dibujable se ocupa de dibujar en pantalla del cliente, cabe mencionar que el cliente este hecho en base a Processing 2 el

cual se ejecuta como un Applet de Java. Al dibujarse los objetos en el cliente se llama a métodos propios del *framework* de Processing para mostrar cuadrados en pantalla, estos tienen un constructor para determinar su ancho, largo y posiciones en un espacio de dos dimensiones, X y Y (Reas, 2007). Sin embargo, para que los objetos puedan pasar del servidor al cliente y se puedan dibujar se ha agregado una referencia a un PApplet, clase en donde se tiene un lienzo para dibujar figuras de Processing. Así mismo para el caso del componente de movible, este se encarga en actualizar la lógica del movimiento que se desee realizar de acuerdo a como el diseño del juego requiera, y de la misma manera lo hace el componente colisionable, pero con la diferencia que usa detección de choque con todos los objetos que posiblemente interactúen.

Con el modelo de objetos de juego expuesto se puede implementar las mecánicas requeridas para el desarrollo de juegos de plataformas. Las mecánicas van a depender más del diseño del juego que del motor, pero es responsabilidad del motor dar control a ese tipo de elementos. Para tener un control entre los objetos y el mundo que puede cambiar de acuerdo al usuario se debe tener un manejador que detecte estos cambios, para esto se ha diseñado un sistema de eventos que se expone en la siguiente sección.

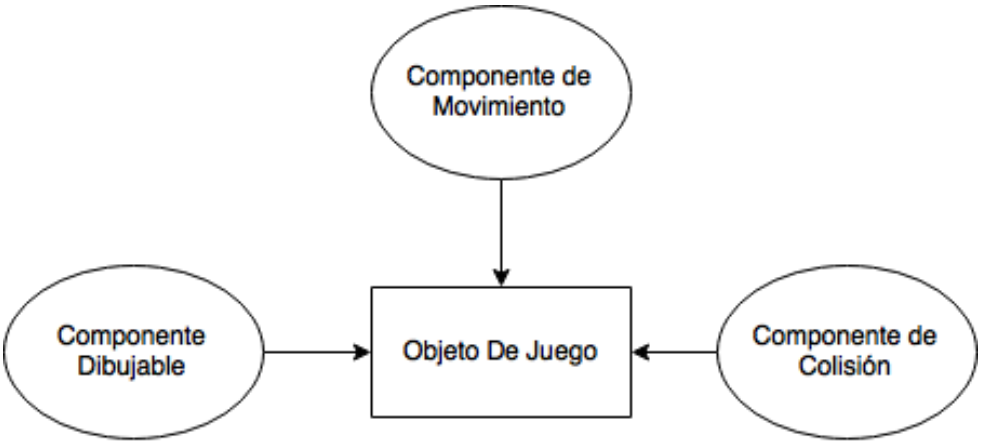

Figura 1:Modelo de objeto de juego por componentes.

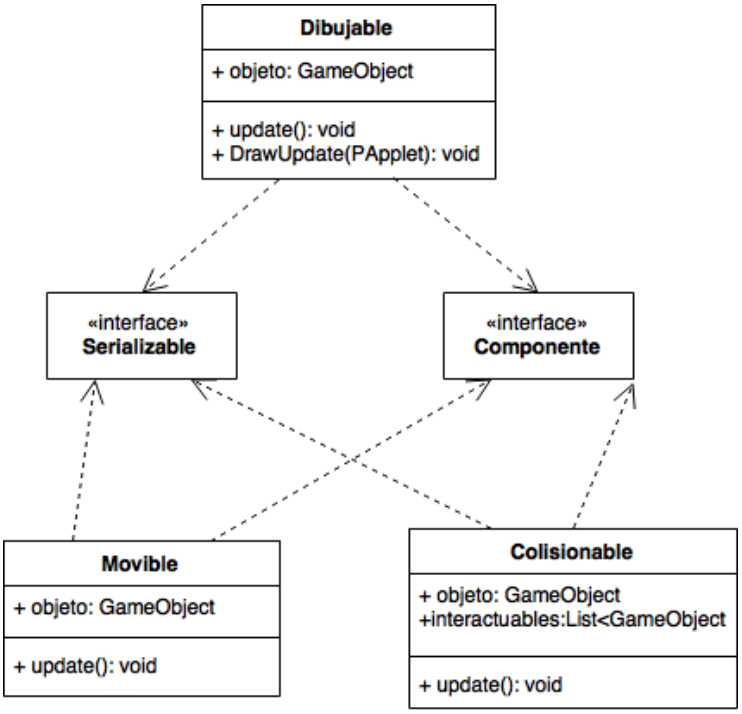

Figura 2:Composición de Interfaces para componentes de objeto de juego.

Sistema de Eventos

Para el control de acciones que realicen los componentes sobre el sistema y la lógica del juego, que el motor ha de controlar, se debe mantener un criterio de priorización pues ciertas acciones pueden ser más importantes que otras. Un ejemplo de priorización de ejecución es la diferencia entre la acción del usuario de pulsar el botón para moverse a la derecha y la de pulsar el botón de salto, entre las dos se llaman al mismo componente , movible, pero las dos toman distintos tiempos de procesamiento, en específico el de movimiento a la derecha sólo le concierne mover al objeto unos pixeles, pero en el caso del salto la lógica es más extensa y el movimiento puede durar varios segundos dependiendo de la distancia recorrida lo que supone mayores cálculos. La mayor prioridad en este ejemplo se le daría al del movimiento a la derecha pues el tiempo es menor y un jugador puede esperar que el movimiento sea más rápido debido a su menor complejidad. Así mismo, el movimiento del salto tendría menor prioridad, pues el usuario puede ver que el movimiento es mucho más grande. Este ejemplo trata de explicar lo que en términos de sistemas operativos se conoce como *Schedulling*, programación de tareas (Silberschtz, 2012). Sin embargo, para representar esto en el motor de juegos se ha decidido poner las tareas a manera de eventos, es así que para el manejo de eventos se ha optado por una arquitectura de tres elementos: eventos, manipuladores (*handlers*) y manejador (*Manager*). Esta arquitectura funciona registrando eventos en manipuladores independientes que se han de ejecutar en el manejador de acuerdo a alguna prioridad (Gregory, 2014).

Para el diseño del sistema de eventos se han tomado algunas consideraciones basadas en conceptos de sistemas operativos como el *scheduling* y sincronización de tareas. La base para una buena programación o *scheduling* de las tareas del sistema es saber el ciclo de ejecución que una tarea ha de tener y la espera que puede aguantar la tarea, además de

asegurar que no se deje al proceso en estado ocioso, en una espera constante o en un punto muerto (Silberschtz, 2012). Las tareas para este sistema se han programado en los manipuladores de eventos. Para evitar los problemas antes mencionados se ha decidido dotar al sistema de eventos con una estrategia de sincronización basada en monitores que consiste en declarar variables cuyos valores definen el estado de la tarea y permiten sincronizar con otros procesos o hilos de acuerdo a estos valores (Silberschtz, 2012).

Por otro lado, las tareas y su ejecución dependen de los tres componentes que tiene el sistema de eventos: Eventos, Manipuladores y Manejador. Para la definición de eventos se ha hecho una enumeración para los casos posibles. Los eventos más comunes para un juego de plataformas son los generados por el jugador por medio de un HID (*Human interaction devise*) que en este caso sería el teclado (Gregory, 2014). Es así como los eventos se han definido por las teclas arriba, abajo, izquierda, derecha y barra espaciadora. Por otra parte, uno de los objetos que se ha implementado en el motor son las plataformas movibles las cuales han de responder a eventos que su enumeración específica también se ha implementado(AUTOMATIC_PATTERN). Así mismo, los manipuladores que son los encargados de ejecutar las acciones que el evento generará. Para representarlos en el sistema se ha hecho una clase llamada `EventHandler` que cuenta con cinco atributos importantes para definir la acción que el objeto ha de hacer y que son necesarios para el *scheduling* del sistema, estos son: `type` (tipo de evento), `timeStamp` (tiempo en el que se registró el evento), `subdivisions` (si el evento va a estar asociado con más eventos), `priority` (prioridad de ejecución) y *a*ge (tiempo de vida máximo que debería esperar para ejecutarse). Además, esta clase cuenta con un método que registra el comportamiento deseado al manejador, y un método para ejecutarlo. Por otra parte, La pieza más importante del sistema de eventos es el manejador pues es el encargado de programar que

manipuladores vayan a ejecutarse y en qué orden de acuerdo a sus propiedades, es decir del *scheduling* en sí. Los atributos de los manipuladores ayudan a que el manipulador decida el orden de ejecución por medio de una cola de preferencia. El manejador ha sido integrado en el servidor pues la ejecución de eventos debe interactuar con los demás objetos de juego y estar sincronizados.

Sin embargo, los eventos no solo deben estar presentes para ser ejecutados, sino que su ejecución debe estar hecha de acuerdo a una lógica para el juego en este caso sería la física para un juego de plataformas. La física en el motor está dada por una estructura de tiempo y lógica en los componentes movible y colisionable. La estructura de tiempo está dada por la clase `TimeLine` y cuenta con un método (`calculateTic()`) y se la usa de tal manera que cada objeto de juego que deba relacionarse con física se actualice haciendo una diferencia de tiempos del procesador obtenido al inicio y al final del lazo de juego. Esta estructura está incluida como propiedad en la clase de Objetos de juego y se usa para actualizar los componentes de Movimiento y Choque. Para el movimiento de objetos que actúan por medio de interacción del usuario, teclado, se ha creado una función para cuando el usuario activa eventos de movimiento de izquierda o derecha en relación al tiempo que está dada por la siguiente ecuación: x=5t/20 y que es positiva cuando el movimiento deseado es a la derecha y negativa cuando es a la izquierda. Así mismo, para eventos de salto se ha hecho un algoritmo basado en el movimiento de caída libre que es el fenómeno que en ausencia de resistencia del aire hace que todos los objetos caigan con la misma aceleración constante (Giancoli, 2008), sin embargo, para mantener simple los cálculos del componente se ha hecho a velocidad constante de y=2t/10. El algoritmo de salto está gobernado por una variable booleana, llamada "jumping", que activa el patrón de movimiento hasta que este se complete al tocar una superficie con propiedades de límites o

choque. La variable que controla el algoritmo es activada cuando el usuario presiona el botón o la tecla de salto. Sin embargo, su funcionamiento puede extenderse por varias iteraciones pues se basa en el lanzamiento de un proyectil: cuando el objeto es puesto en el salto inicia con una velocidad constante hasta que llega a su punto máximo que es tres veces mayor que el de su propia altura, después de esto el objeto ha de empezar su descenso con la misma velocidad hasta llegar al borde o chocar contra otro objeto. EL comportamiento de movimiento muchas veces va a depender del de choque pues para varios objetos en el juego la condición de choque va a ser obligatoria (saltar sobre plataformas, por ejemplo), es por esto que en patrones como el de salto es necesario saber si ocurrió una colisión en la parte de abajo del objeto para comprobar si el movimiento continúa o se para.

Por otro lado, el manejador ha de ser un árbitro para el sistema y quien intervenga entre la lógica del servidor y los manipuladores que vienen desde el cliente. El sistema está hecho de tal manera que los manipuladores se han de generar en los clientes, luego han de llegar al servidor quien tiene al manejador de eventos corriendo en paralelo con un hilo que se encarga de llenar una cola de prioridad con manipuladores para efectuar operaciones de sincronización entre la lógica que maneja el servidor y la ejecución de los manipuladores. Una cola de prioridad es una estructura de datos que permite hacer *schedulling* en su colección de objetos y puede estar ordenada por la prioridad más alta a la más baja, pero para que los objetos puedan ser comparados deben tener una interfaz comparable (Weiss, 2012). La cola de prioridad implementada para el proyecto tiene la particularidad de compartir elementos tanto en la lógica principal del servidor como en el hilo de ejecución por esto debe ser atómica. En consecuencia, se ha usado una cola de prioridad del tipo *PriorityBlockingQueue* que es una colección atómica de Java (Oracle, 2016). Por otro lado,

para que la cola de prioridad ejecute los modificadores y que estos tengan efecto dentro de los objetos de juego del servidor, se debe sincronizar las operaciones realizadas en el manejador con el hilo principal del servidor. Si bien es cierto los manipuladores al estar dentro de una colección atómica no deberían sufrir problemas de sincronización, si estos son ejecutados en un hilo sin esperas el sistema podría caer en un lazo infinito que en ausencia de eventos consumiría recursos del procesador infinitamente. Para no tener este problema se ha usado la estrategia de sincronización por medio de un objeto monitor que cambia su estado a espera cuando la cola está vacía y cuando la cola está llena cambia de estado para permitir que se continúe con la ejecución del hilo del manejador.

El sistema de eventos descrito permite que la lógica se comparta entre cliente y servidor, sin embargo, el sistema que se encarga de la comunicación usa otros principios computacionales. Como ya se ha descrito los manipuladores necesitan *scheduling* para funcionar, sin embargo, para la comunicación es necesario tener una manera parecida si es que se va a intercambiar información de eventos y de objetos que como se describió anteriormente son tipos distintos que manejan cosas diferentes del motor.

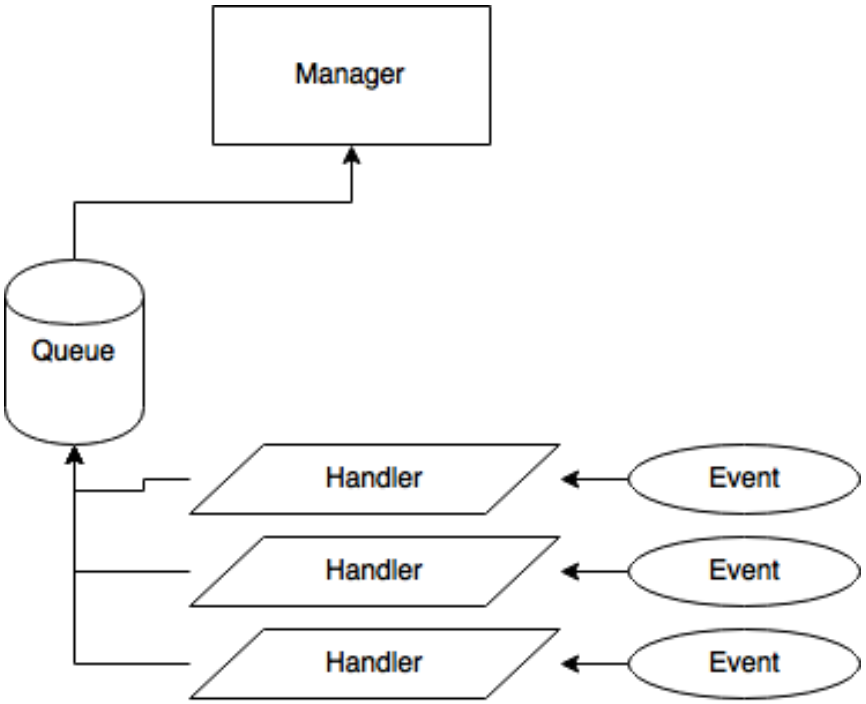

Figura 3:Diagrama del Sistema de Eventos.

Arquitectura de red para múltiples jugadores

Para la arquitectura de red se ha optado por el modelo de cliente servidor pues permite que el sistema trabaje de una manera distribuida y que la experiencia de usuario sea enfocada para juegos con varios jugadores. En particular el modelo cliente servidor en motores de juego requiere resolver la pregunta de si se desea que el cliente sea inteligente, o un simple presentador de imágenes (Roberts, 2015). Es por esto que para este proyecto se ha decidido por un punto intermedio en el cual el cliente recibe datos pre procesados por el servidor para mostrarlos en pantalla, y además genera y envía controladores de eventos al servidor para su ejecución correspondiente.

Para la implementación del modelo de cliente servidor se debe tener claro cómo es que este funciona y qué estrategia se tendrá para el desarrollo de todos sus componentes. El modelo de arquitectura de cliente-servidor es un modelo de sistema distribuido en el cual se muestra como los datos y los procesos son distribuidos entre una serie de procesadores o computadoras. Los componentes de esta arquitectura son: un conjunto de servidores autónomos que den servicios a otros subsistemas, un conjunto de clientes que llamen a los servicios ofrecidos por los servidores, y una red que permita que los clientes accedan a los servidores (Sommerville, 2000). De esta manera se requiere un servidor que sea capaz de mantener conexiones con varios clientes, dicho servidor además de ser capaz de lograr los objetivos de un motor de juegos, debe mantener la lógica del juego y comunicarlo sincronizadamente a todos los clientes. Para realizar esto se ha implementado una solución con enfoque en selectores de múltiples canales, siendo cada canal una conexión a un cliente en particular. Por otro lado, los clientes se han hecho en base a Processing, versión 2, que es un *framework* hecho sobre Java orientado a la programación de artes visuales (Processing,

s.f.), de esta manera se asegura que el manejo de graficas del sistema esté a cargo del *framework* al implementarlo en los clientes.

Así mismo, para el diseño del sistema es importante entender la red en la que se ha de usar la arquitectura, y el objetivo que para este caso es permitir el desarrollo de juegos en redes de área local con protocolo Ethernet. Una red de área local o LAN es una red de propiedad privada que opera dentro y cerca de un solo edificio como una casa, oficina o una fábrica, además son muy usadas para conectar computadoras personales y aparatos electrónicos para que estos puedan intercambiar y compartir recursos e información (Tanenbaum, 2011). Con esto en mente, el resultado del motor permitiría a varias computadoras conectarse para que múltiples jugadores puedan disfrutar de una partida de juego. Sin embargo, para que las computadoras conectadas en red puedan enviar y recibir paquetes se necesita de protocolos que permitan que un paquete de datos sepa a qué computadora de la red ir, entonces se ha optado por el uso de los siguientes protocolos: Ethernet para la capa de enlace, IP para la capa de enlace y TCP para la capa de transporte, cabe mencionar que la capa de aplicación ha de estar dada por los programas del servidor y del cliente.

Para implementar esta arquitectura se han hecho varias pruebas con diferentes enfoques, el primero por medio de server sockets y el segundo por medio de *socket channels* con selectores. Según la documentación de Oracle, un socket en java es un punto final de una conexión de comunicación entre dos programas que se ejecutan en la red que además usa TCP como capa de transporte. La diferencia entre los *server sockets* y los *socket channels* es la capacidad de usar selectores por canales.

El enfoque por Server Sockets trata de múltiples conexiones multihilo con varios sockets conectados a otros varios server sockets para dos puertos, uno para intercambio de

objetos de juego y otro para recepción de eventos, pero que ha resultado ineficiente. Un Server Socket es una clase de java que espera por peticiones que vienen de la red, realiza operaciones basadas en ese pedido y que puede enviar algún resultado al solicitante del pedido. Es así que para este enfoque en particular se ha usado dos objetos Server Sockets que hacen operaciones *accept* (operación para aceptar una conexión entrante) en el hilo principal. Después de cada *accept* el sistema llama a un hilo, uno para intercambio de objetos y otro para ejecución de eventos. El hilo de intercambio de objetos se ha de sincronizar con el de ejecución de eventos para que de esta forma se envíen objetos actualizados al cliente y se mantenga la lógica del juego. Sin embargo, este enfoque genera hilos por cada conexión aceptada en el socket server, y también nuevos sockets para enviar información a quien hizo la petición, lo que lleva a un congestionamiento de la red y uso excesivo del procesador. Cabe mencionar que se han hecho pruebas en dos tipos de procesadores: AMD A8 y en un Intel Core i7, y el comportamiento del motor fue totalmente diferente entre ambos pues en el AMD A8 la carga de procesamiento hacía que el programa y los demás procesos de la computadora sean más lentos, sin embargo en el caso del Intel i7 el programa se ejecutaba fluidamente, pero de igual manera congestionar la red interna de la computadora.

Por otro lado, el enfoque por *Socket Channels* usa un socket que funciona como selector en el servidor el cual recibe conexiones entrantes y las registra en canales. Los clientes van a tener tres tipos de operaciones básicas para conectarse con el servidor: Lectura, escritura y de aceptación. Cada cliente, dentro de su lazo principal, realizará ordenadamente los siguientes pasos: primero, enviar al servidor una lista de objetos serializables con el objeto de juego perteneciente al jugador, y con eventos que hayan surgido dentro del cliente como eventos provocados por el teclado. Después, el cliente

esperará a recibir una lista con los objetos de juego actualizados por el servidor. De esta manera el servidor tiene que ajustar los canales para que lean y escriban en relación con lo que el cliente quiere hacer. Para lograr esto se ha decidido alternar las operaciones del canal de la siguiente manera: La operación de aceptación se da cuando la conexión es nueva y por primera vez por parte del cliente y del servidor. Es por esto que cuando el servidor detecta una nueva conexión se ha de registrar un canal que sea capaz de recibir datos del cliente, en otras palabras se habilita a la conexión para recibir datos y que el servidor pueda leer la información que ha de mandar el cliente. Así mismo, una vez que el cliente haya escrito datos en el servidor, este ha de cambiar el canal para que pueda enviar datos del servidor al cliente. De esta manera se asegura que las operaciones de lectura y escritura se mantengan sincronizadas cada que se presenten nuevas solicitudes desde el cliente. Por parte del rendimiento en computadoras, en las que se probó esta arquitectura con un solo jugador en localhost , el resultado es invariable y satisfactorio pues han presentado un juego fluido sin alterar o ralentizando otros procesos.

El sistema de red para este proyecto es uno de los pilares importantes. Como objetivo del sistema se ha dado el hecho de que este se pueda manejar en red con varios jugadores conectados y para evaluar la experiencia que estos tengan al conectarse se debe medir los requerimientos de uso de las computadoras que tendrán corriendo a los procesos. Es así que una vez que el sistema de red es implementado en el motor de juego, se ha procedido a monitorear el rendimiento del sistema en conjunto.

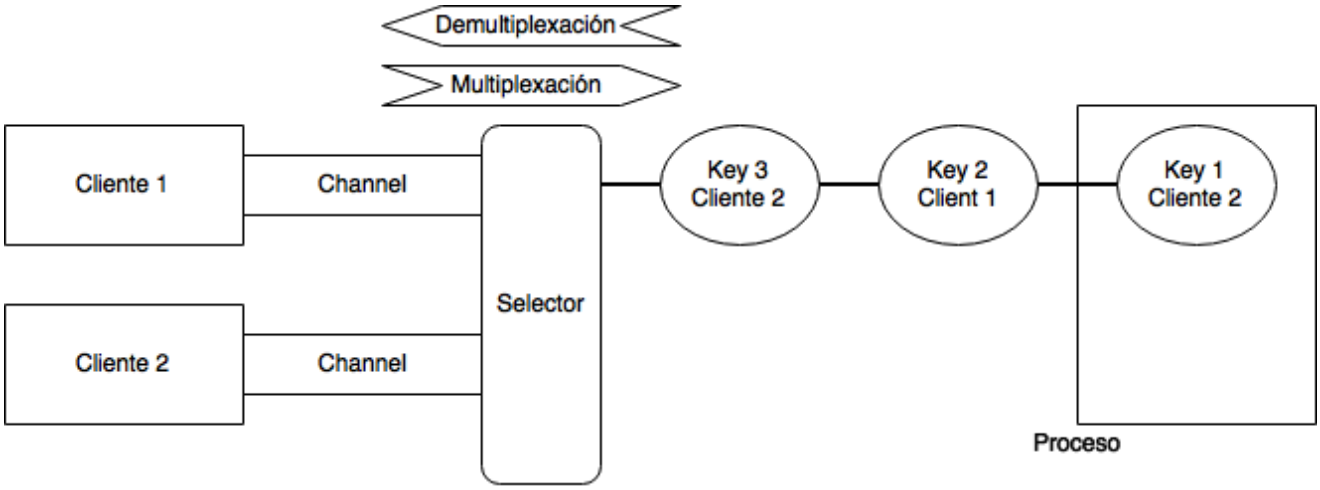

Figura 4:Diagrama de sistema de Red

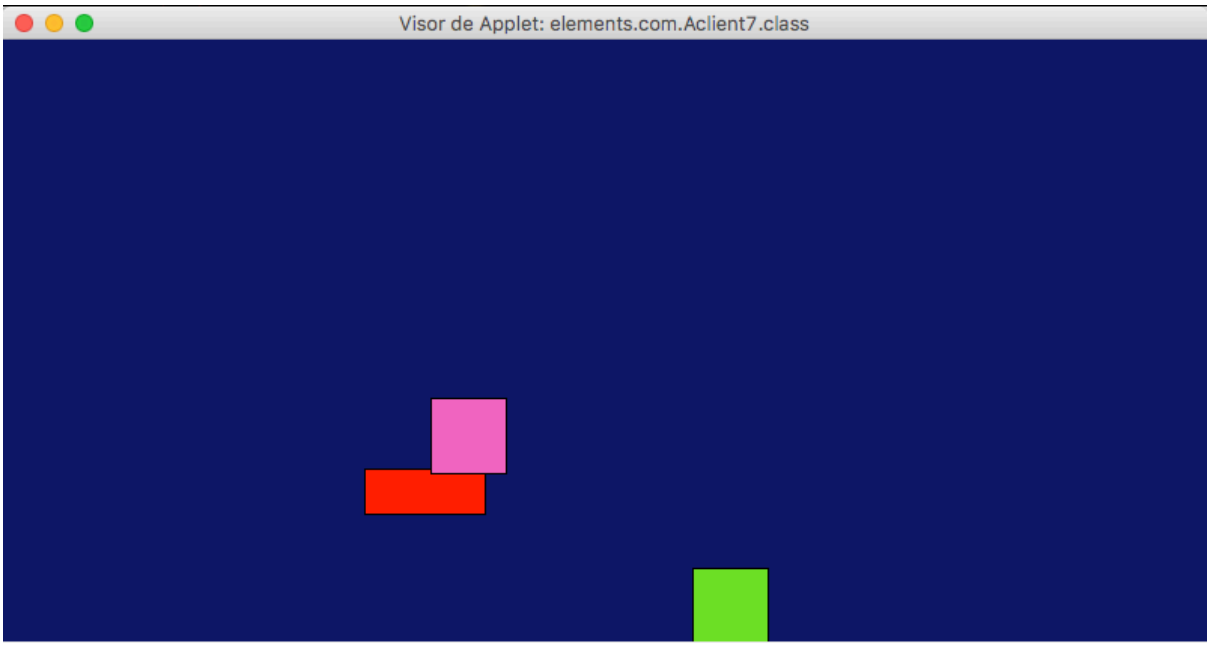

Figura 5: Juego funcional hecho para probar el sistema

Análisis del rendimiento del sistema.

Para medir el rendimiento del sistema se ha obtenido datos del uso del CPU, memoria y de los paquetes de red resultantes de pruebas hechas con un juego funcional que prueba los aspectos básicos del motor. Como el sistema está hecho para funcionar en red se lo ha probado de dos maneras, la primera por *localhost* y la segunda por LAN. Las pruebas por *localhost* buscan demostrar que el motor funciona para la creación de juegos de un solo jugador, por otra parte las pruebas en LAN buscan asegurar una experiencia multijugador para el motor.

Tanto las pruebas de *localhost* como en LAN fueran hechas con ayuda de hilos que recolectaban el uso del CPU y la memoria usada por el proceso, de esta manera se recolectaron datos para obtener conclusiones de su funcionamiento. Así mismo, para la recolección de datos de red se ha usado Wireshark para ver la cantidad y el tamaño de los paquetes para hacer un análisis en función del tiempo. Cabe mencionar que los datos obtenidos usan diferentes unidades: Para el uso del CPU se usan porcentajes, para el uso de memoria se usa Bytes y para el análisis de red se ha usado paquetes/segundo, todos ellos analizados versus el tiempo siendo los datos obtenidos de los hilos en un intervalo de aproximadamente 100 milisegundos y los obtenidos por Wireshark de intervalos de 1 segundo. Cabe mencionar que cada prueba se ha realizado con tres experimentos para luego obtener un promedio que resuma el comportamiento de los experimentos.

Pruebas en Localhost.

Estas pruebas se han hecho en una misma computadora que funciona al mismo tiempo de servidor y de cliente, con aproximadamente tres minutos de juego con envío de eventos aleatorios según el uso de un jugador humano. Cabe mencionar que la computadora tiene un procesador de Intel Core i7 a 2.8 GHz con 4GB de RAM. Por otra parte, se han hecho dos tipos de pruebas para *localhost*. La primera consiste en uso de un solo jugador y la segunda uso multijugador, varios clientes en *localhost*.

Para la prueba de un solo jugador se ha encontrado una línea de tendencia de y=-0.0022x+4.213 para el uso del procesador en el servidor, mostrando así un gradiente decreciente que demuestra que el uso del CPU es mayor al inicio del programa del servidor. Por otro lado, la tendencia del uso del procesador para la aplicación del cliente fue de y=-0.0037x+14.236, mostrando un comportamiento similar, pero con un uso más alto que el del servidor. En el caso del análisis de memoria para un solo jugador, el servidor ha mostrado una tendencia a crecer de y=730.13x+1E7 y en el caso del cliente una tendencia de y=698.61x +2E7, con la particularidad que para los dos casos el uso del recolector de basura aumenta entre el intervalo 400 y el intervalo número 1000. Como resultado del análisis de red de un solo jugador se ha detectado una tendencia del número de paquetes enviados por segundo de y=0.0225x+203.26 que muestra un leve crecimiento de paquetes enviados al avanzar el juego y un promedio de paquetes enviados de 205 paquetes/segundo. Por otro lado se ha detectado que el 50.01% de paquetes tenían un tamaño de entre 40 a 79 bytes que pertenecen a paquetes tipo ACK, y 49.99% de los paquetes fueron de entre 1280 a 2559 bytes que contenían datos de objetos de juego y eventos.

Así mismo para la prueba de multijugador en *localhost* se obtuvo resultados similares a las de un solo jugador, pero con líneas de tendencia más empinadas para la memoria y CPU y los clientes. La tendencia para el uso del CPU en el servidor ha sido de y=-0.0019x+4.7843, que igual que en el caso anterior, muestra un decrecimiento de procesamiento desde el inicio del programa, siendo los primeros 200 intervalos poseedores de porcentajes de uso mayores que el resto sobrepasando el 20% del CPU, pero el resto del tiempo el uso del CPU se estabiliza en un promedio de 3% aproximadamente. Así mismo, el uso de CPU para los clientes, dos, en *localhost* ha mostrado una tendencia de y=-0.0029x+13.564, que igualmente muestra un decrecimiento desde que el programa empieza a ejecutarse en los primeros 100 intervalos con picos de hasta el 31% de uso del CPU, para luego mantener un 11% de promedio en uso del CPU, 8 puntos más arriba que el servidor. Por otra parte, el uso de memoria para el servidor fue de y=751x+1E7, tendencia que se encuentra muy parecida a la obtenida para un solo jugador, así mismo por parte de los clientes se obtuvo una tendencia de y=1406.5x+2E7 que es mayor que la tendencia para un solo jugador, lo cual supone que cuando en el sistema interactúan más objetos, la memoria necesaria aumenta en los clientes. Finalmente, en el análisis de red se ha obtenido una tendencia de crecimiento mayor a la de un sólo cliente, que resulta en y=0.1375x+319.02con promedio de 333 paquetes enviados en un segundo, promedio el cual se mantiene como tendencia en la vida del sistema , sin embargo la ecuación resultante es una línea que crece con pendiente positiva que se explica porque al inicio de la vida del servidor se envían pocos paquetes tipo ACK hasta que se realizan las primeras conexiones.

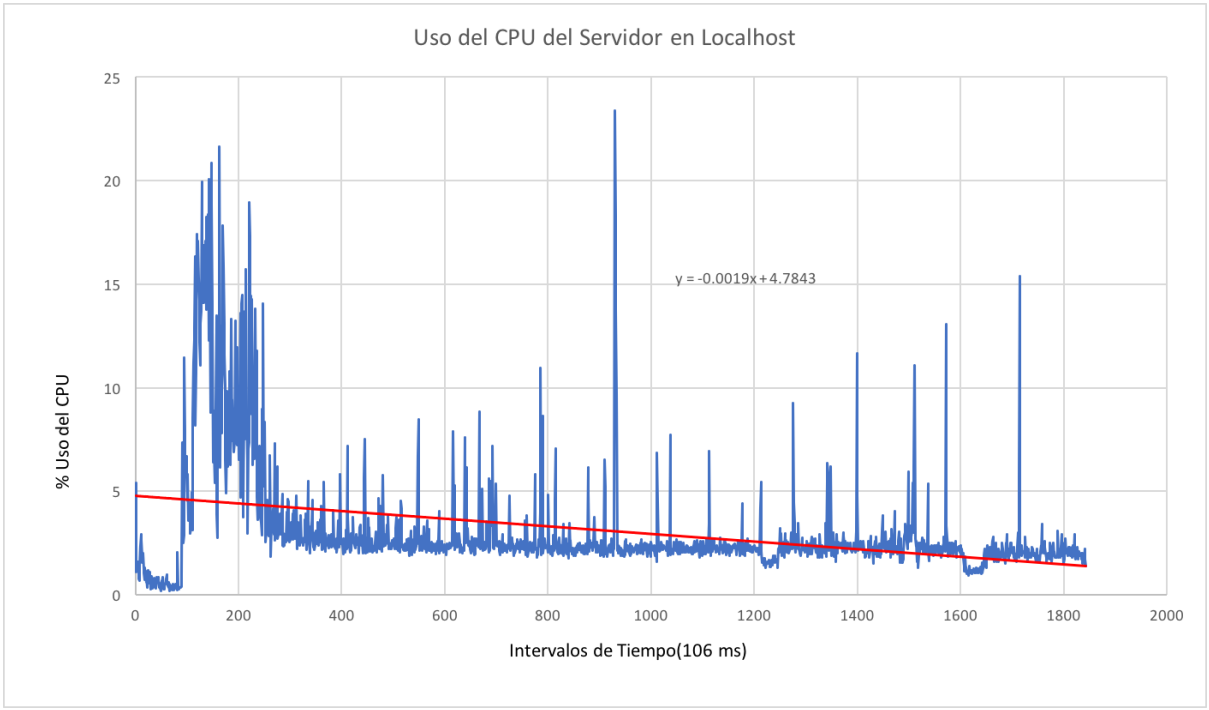

Figura 5: Uso del CPU del Servidor en localhost

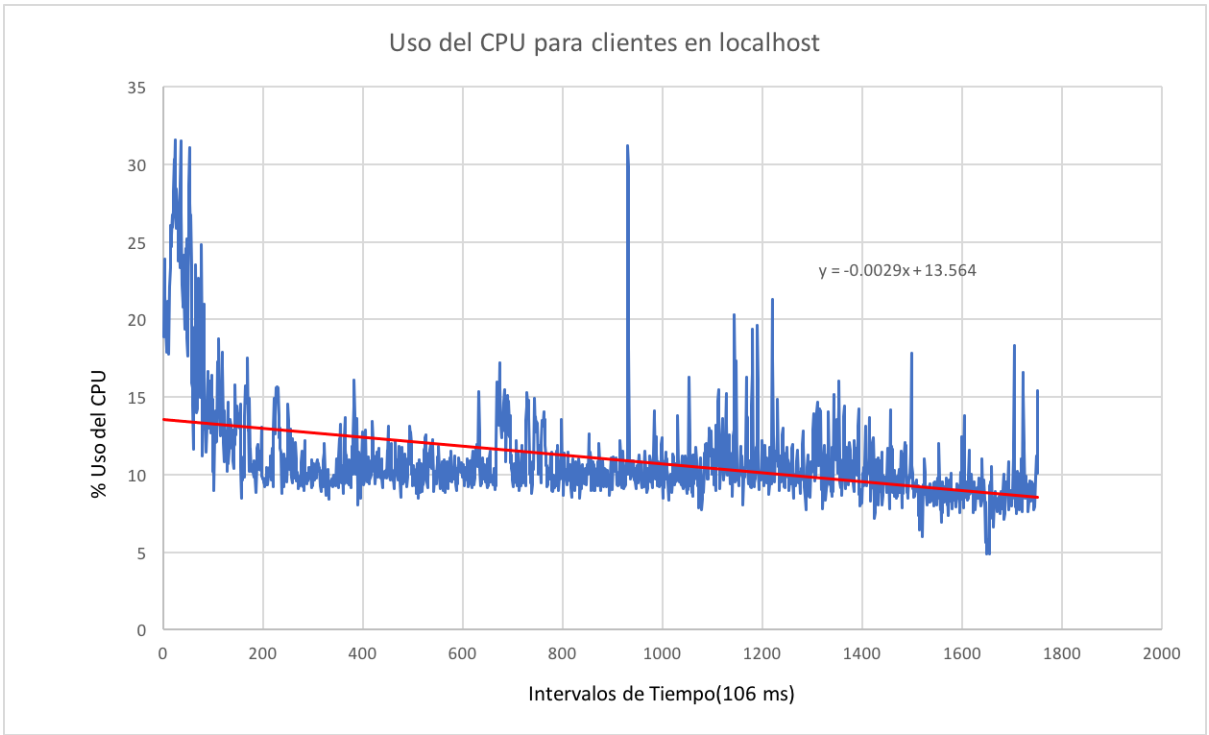

Figura 6: Uso del CPU para clientes en localhost

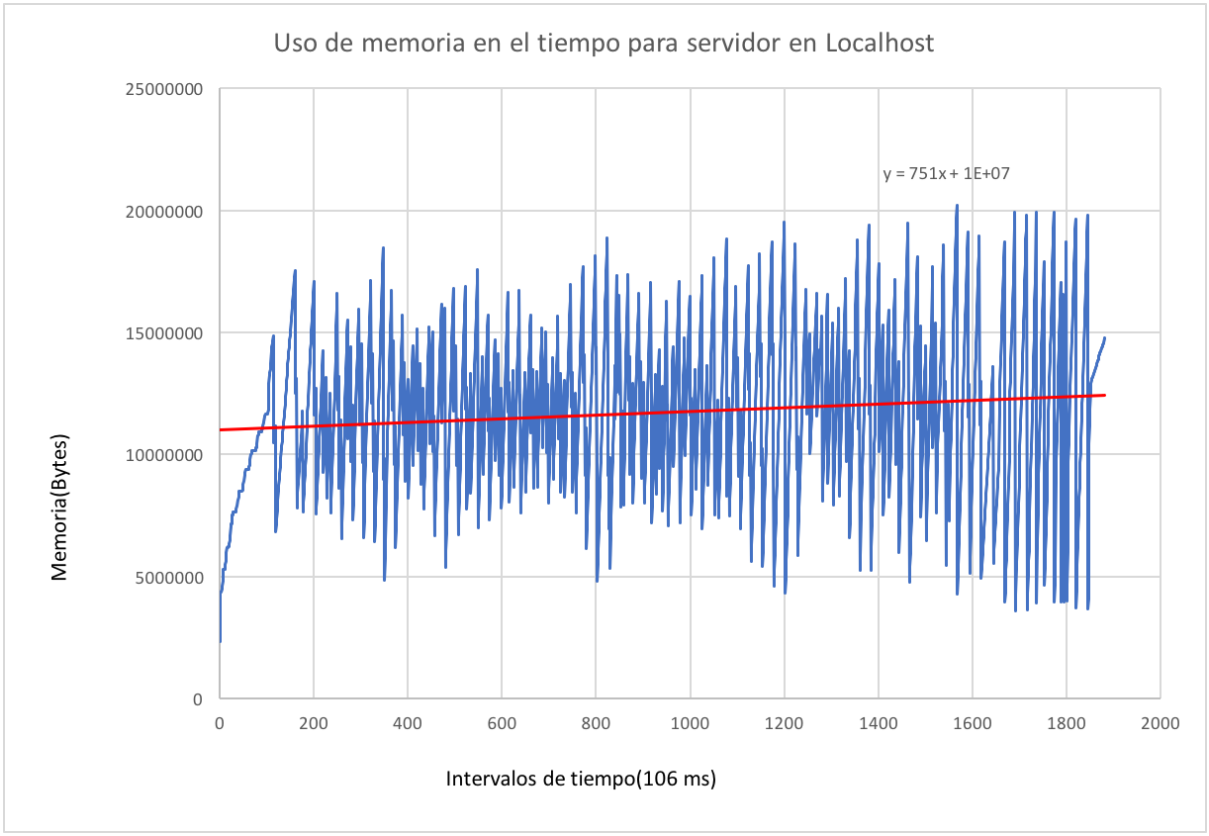

Figura 7: Uso de memoria en el tiempo para servidor en localhost

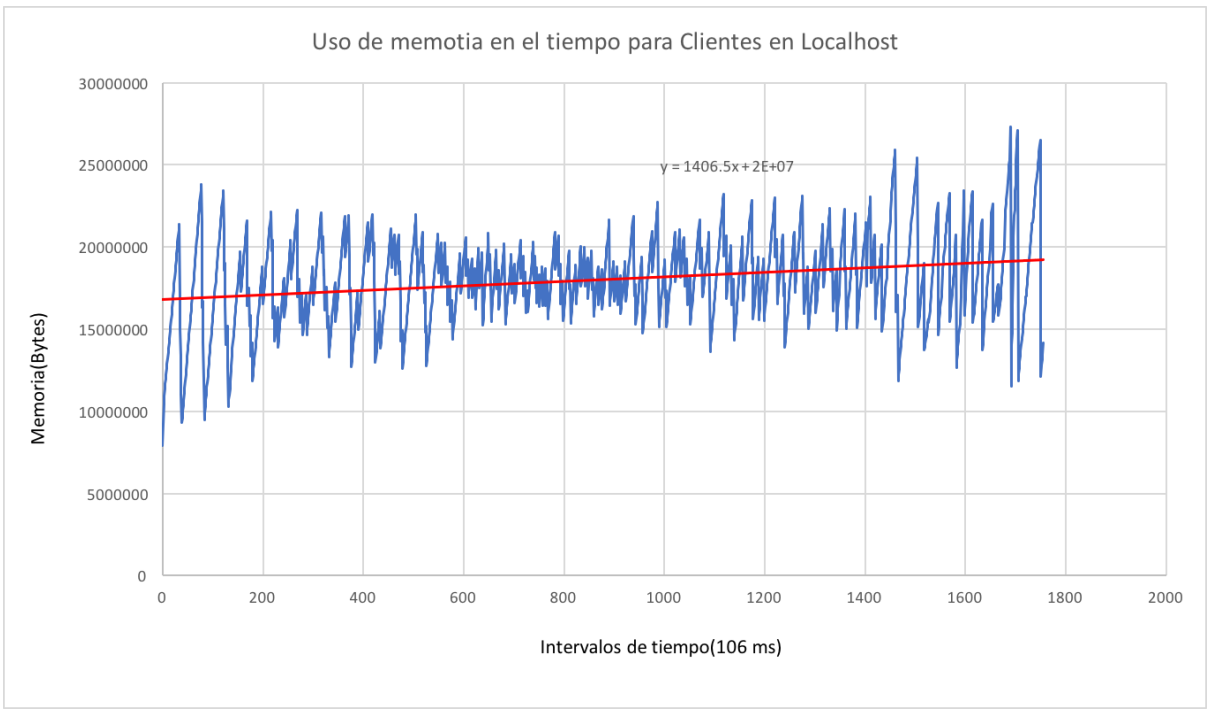

Figura 8: Uso de memoria en el tiempo para clientes en localhost

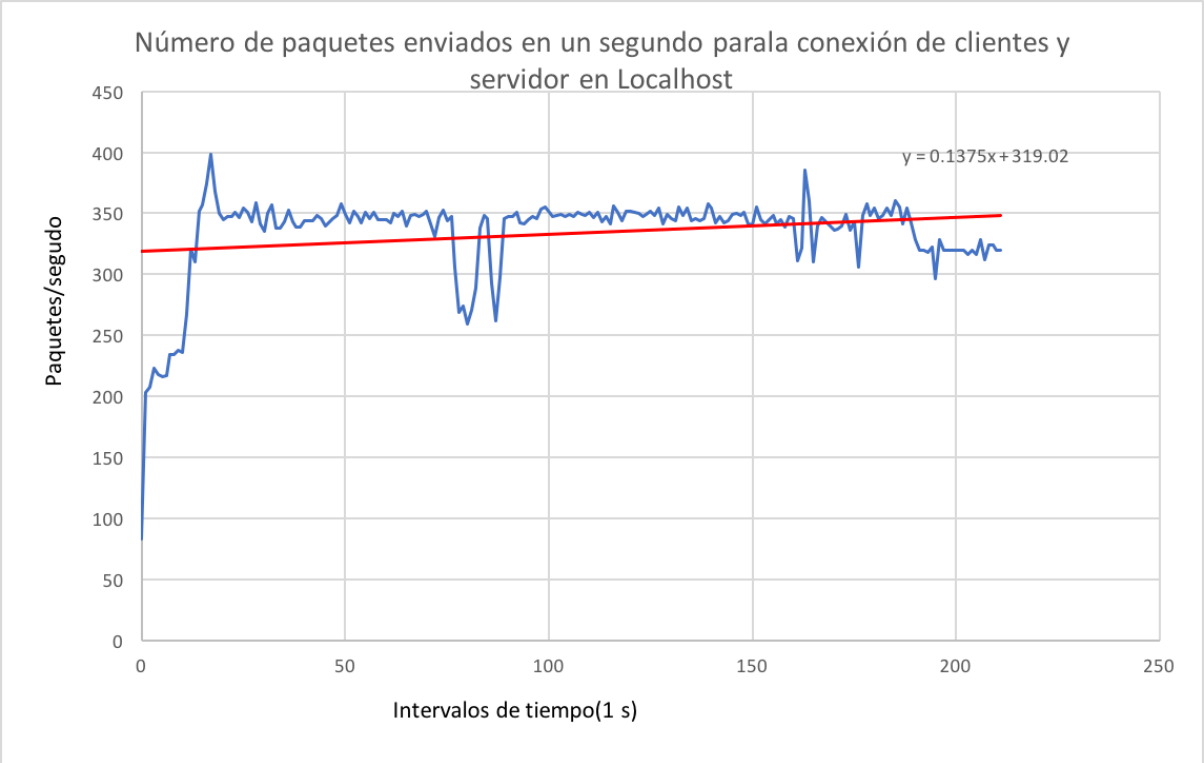

Figura 9: Número de paquetes enviados en un segundo para la conexión de clientes y servidor en localhost

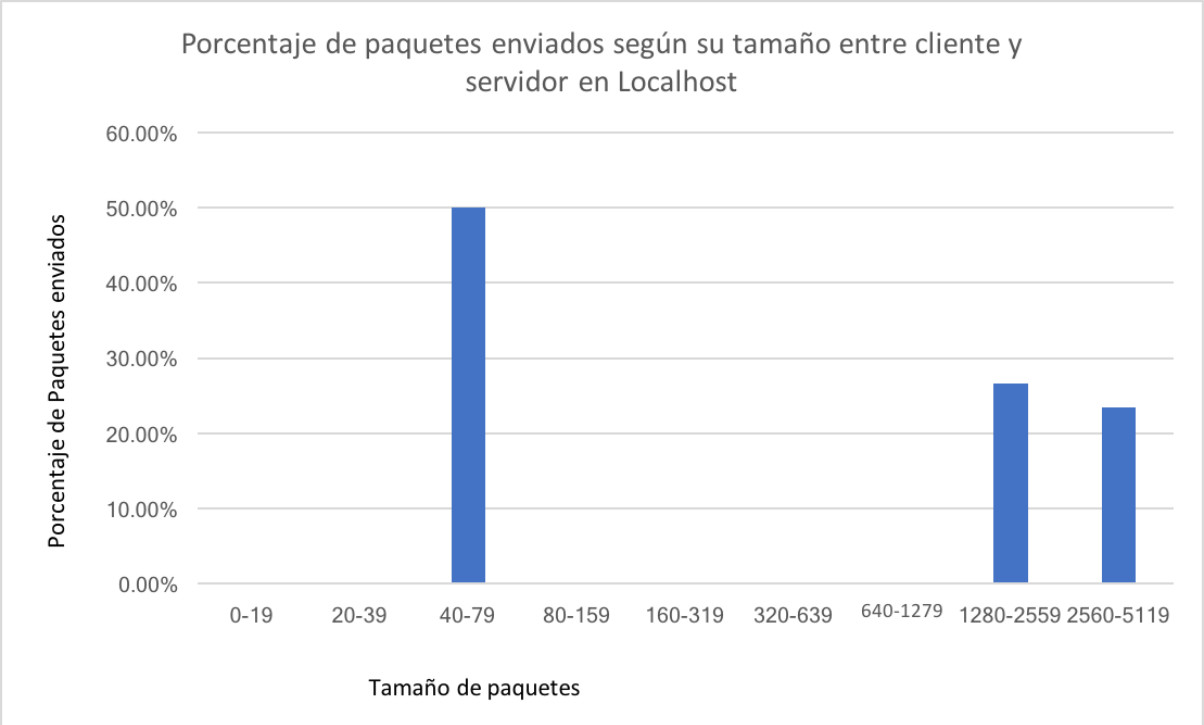

Figura 10: Porcentaje de paquetes enviados según su tamaño entre cliente y servidor en localhost

Pruebas en LAN

Para este tipo de pruebas se ha usado la computadora antes mencionada, para el experimento en *localhost*, como servidor y además un equipo para generar una red LAN por cable Ethernet de 10 Mbps de ancho de banda. Y como en el experimento anterior, se han hecho pruebas con uno y con dos jugadores conectados al servidor.

Para el experimento con un sólo jugador conectado en LAN al servidor se ha encontrado una tendencia de y=-0.0012x+4.2814que es aproximadamente igual a la obtenida para los experimentos de *localhost*, además de presentar el mismo promedio de 4% al estabilizarse el uso del CPU después del intervalo número 200. Así mismo, el cliente ha mostrado una tendencia de uso de CPU similar al del experimento anterior con una tendencia de y=-0.0043x+13.047y con un promedio de aproximadamente 13% de uso del CPU después del intervalo 200. Sin embargo, en el caso de la memoria utilizada el servidor experimenta un incremento a partir del intervalo 1500 de un máximo de 14 Mb a 19 Mb, sin embargo por causa del recolector de basura de java la línea de tendencia ha sido de y=522.02x+1E7que es menor que la obtenida para *localhost* pero en este caso el uso de memoria es menos uniforme que las anteriores. En el caso de la memoria de los clientes la memoria sigue una tendencia más uniforme, pero con un máximo de 28 Mb, mayor que en el servidor, y con ecuación de tendencia de y=1249.7x+2E7. Por otra parte, en el análisis de red se encontró una tendencia de y=0.031x+255.68con un promedio de 355 paquetes por segundo y con un 33.33% de paquetes de tamaño de entre 1280 a 2559 bytes que representan la segunda mayoría después de un 33.34% de paquetes de ACK, mostrando así que en conexión ethernet se envían paquetes más grandes que en *localhost*, esto puede ser porque el protocolo es más complicado que en *localhost*.

En el experimento de LAN para múltiples jugadores se han usado dos computadoras como clientes y un servidor que es la misma computadora usada anteriormente. Las computadoras que han sido usadas como clientes tienen diferentes procesadores, la primera usa un Intel Core i7 a 2.9GHz con 16Gb de memoria RAM, la segunda usa… Cabe mencionar que este experimento fue el que con más datos se han contado. Para el análisis de uso del CPU del servidor se encontró mayor número de picos que en cualquier otra prueba y un máximo alcanzado en media ejecución de 28% de uso del CPU, sin embargo sigue una tendencia de y=-0.00017x+6.4801 con un promedio de 6.4% de uso es la prueba en donde el uso del CPU se ha duplicado en relación a las anteriores y aunque es decreciente el número de picos es mayor que en otras pruebas. En el caso del CPU de los clientes la tendencia es similar a la de anteriores casos de y=-0.00057x+11.54 que muestra casi una constante de 11.5% de uso, esto hace suponer que el rendimiento del cliente puede estar dado por la estructura que está hecho Processing más que de las particularidades del motor propuesto. En el caso de la memoria del servidor se ha notado un patrón que diferencia de los anteriores experimentos y es que el uso del recolector de basura es más común que en los anteriores pero con una tendencia a crecer según la ecuación de y=987.58x+1E7, pero en el caso de los clientes el llamado del recolector de basura es menos frecuente que en anteriores experimentos y a diferencia de los anteriores la memoria requerida tiende a bajar por la tendencia y=-1494.9x+2E7.Por otra parte, el análisis de red ha mostrado que el número de paquetes enviados por segundo es mayor que en los anteriores pues está dado por la tendencia de y=0.2259x+589.62, además de que el 38.63% de paquetes han tenido un tamaño de entre 1280 a 2559 bytes, representando la mayoría de información intercambiada en el sistema

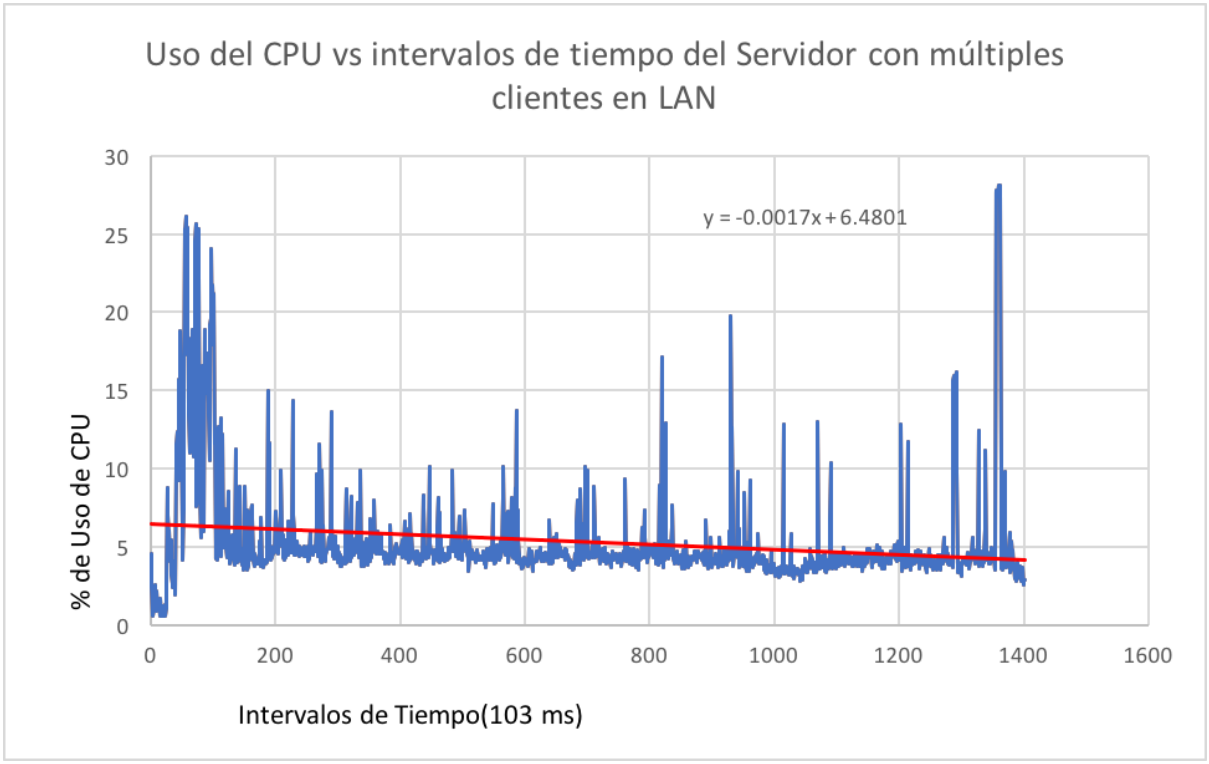

Figura 11:Uso del CPU vs Intervalos de tiempo del servidor con múltiples clientes en LAN

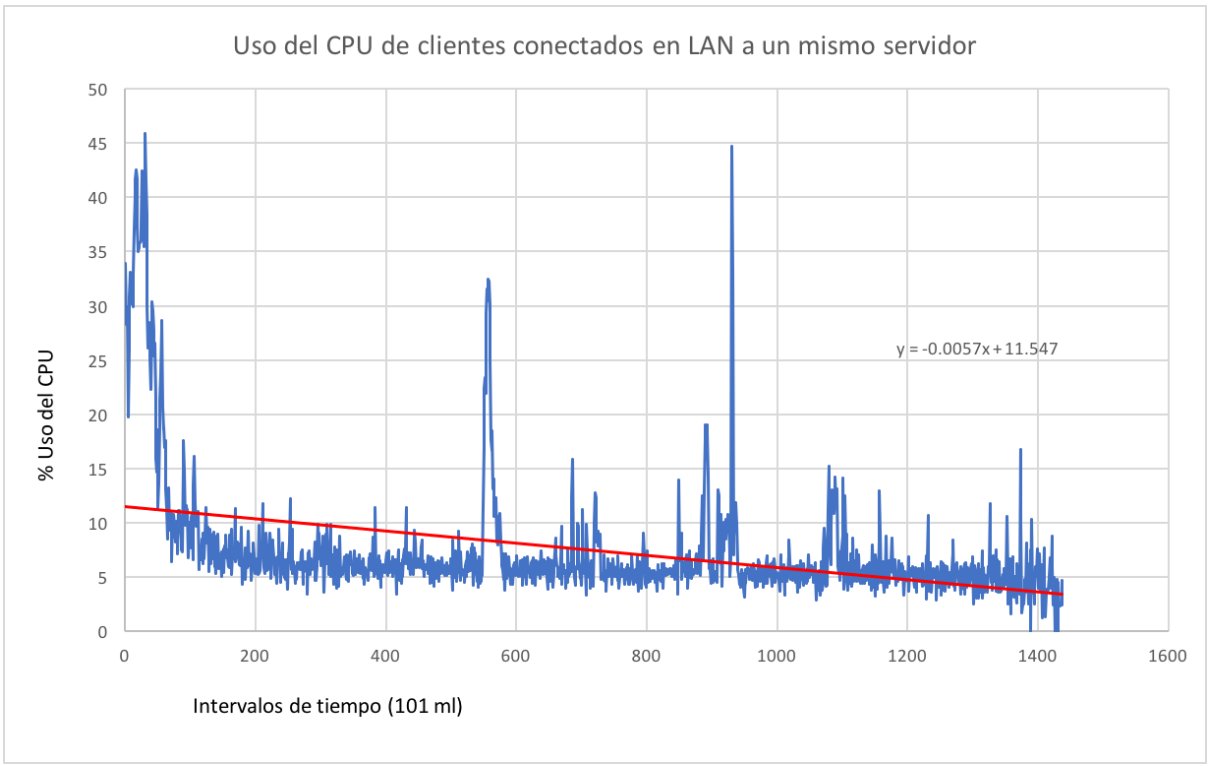

Figura 12: Uso del CPU de clientes conectados en LAN a un mismo servidor.

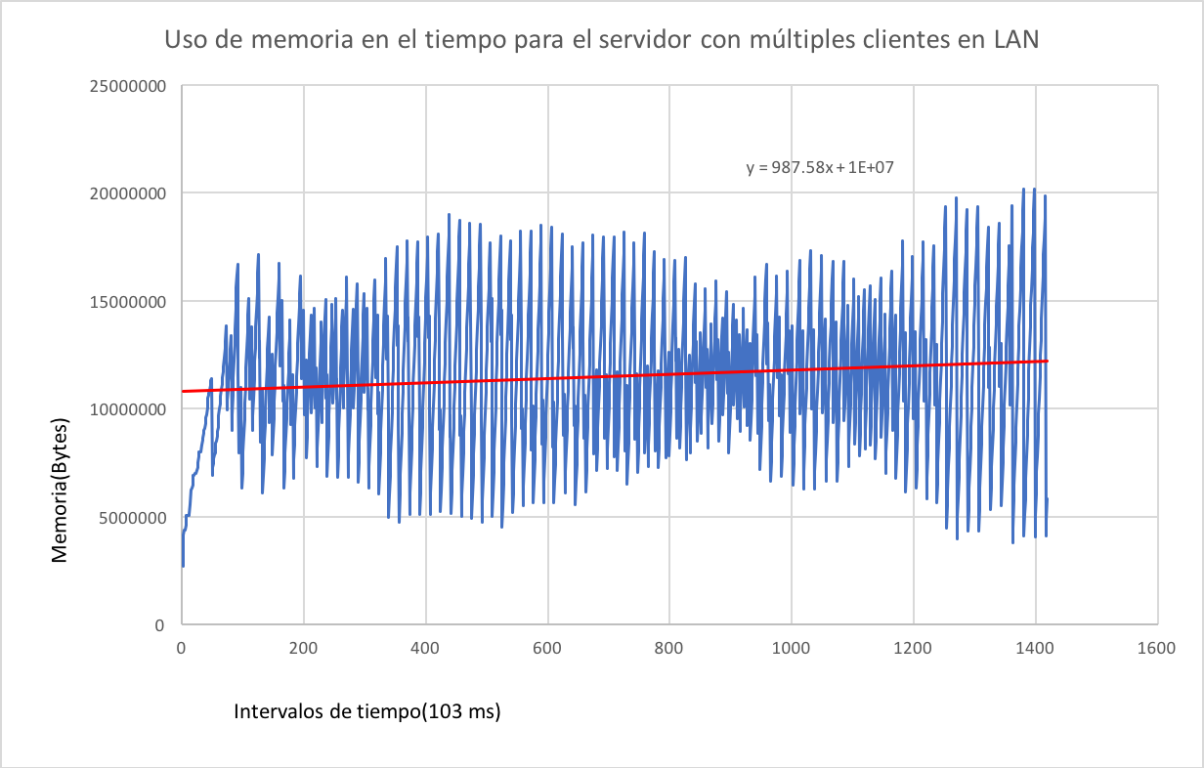

Figura 13: Uso de memoria en el tiempo para el servidor con múltiples clientes en LAN.

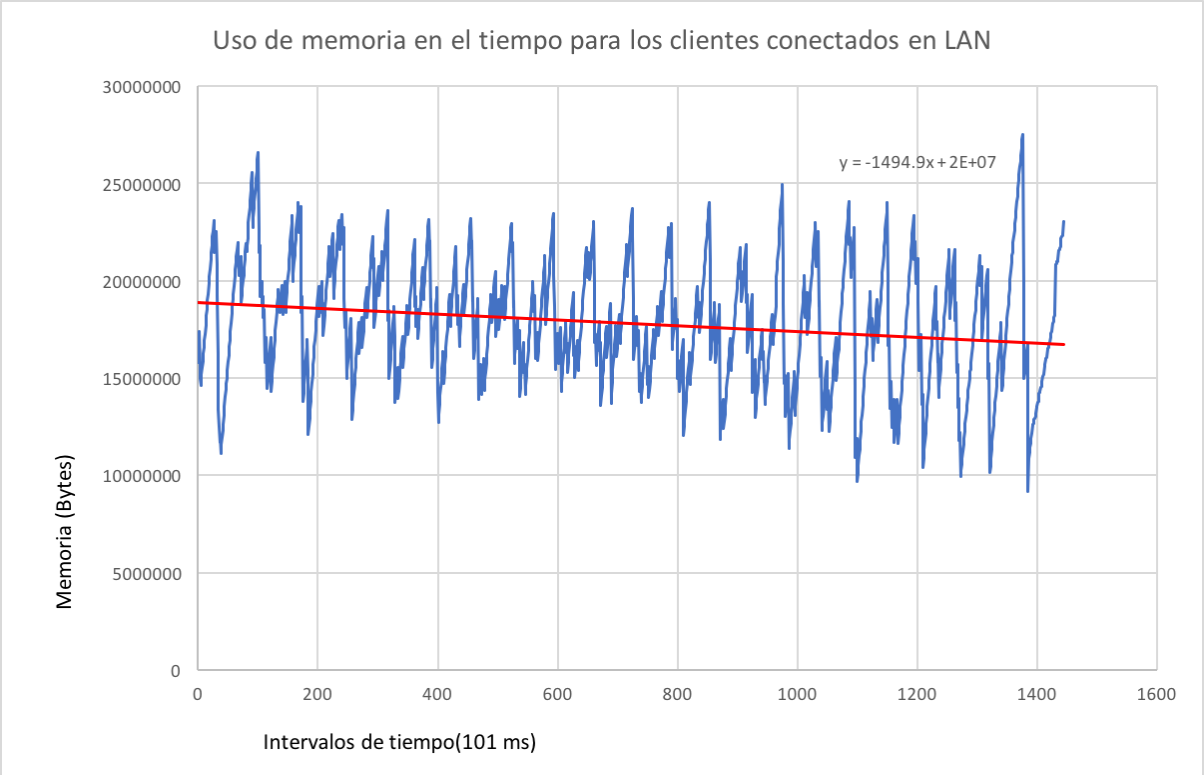

Figura 14: Uso de memoria en el tiempo para los clientes conectados en LAN.

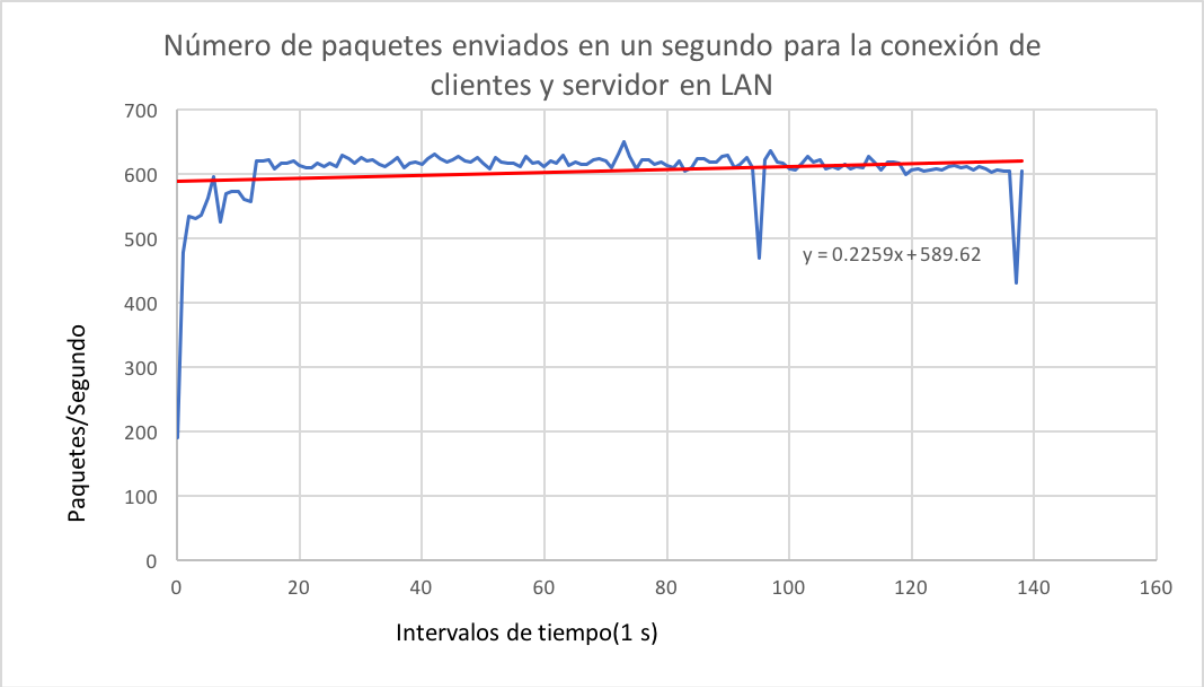

Figura 15: Número de paquetes enviados en un segundo para la conexión de clientes y servidor en LAN.

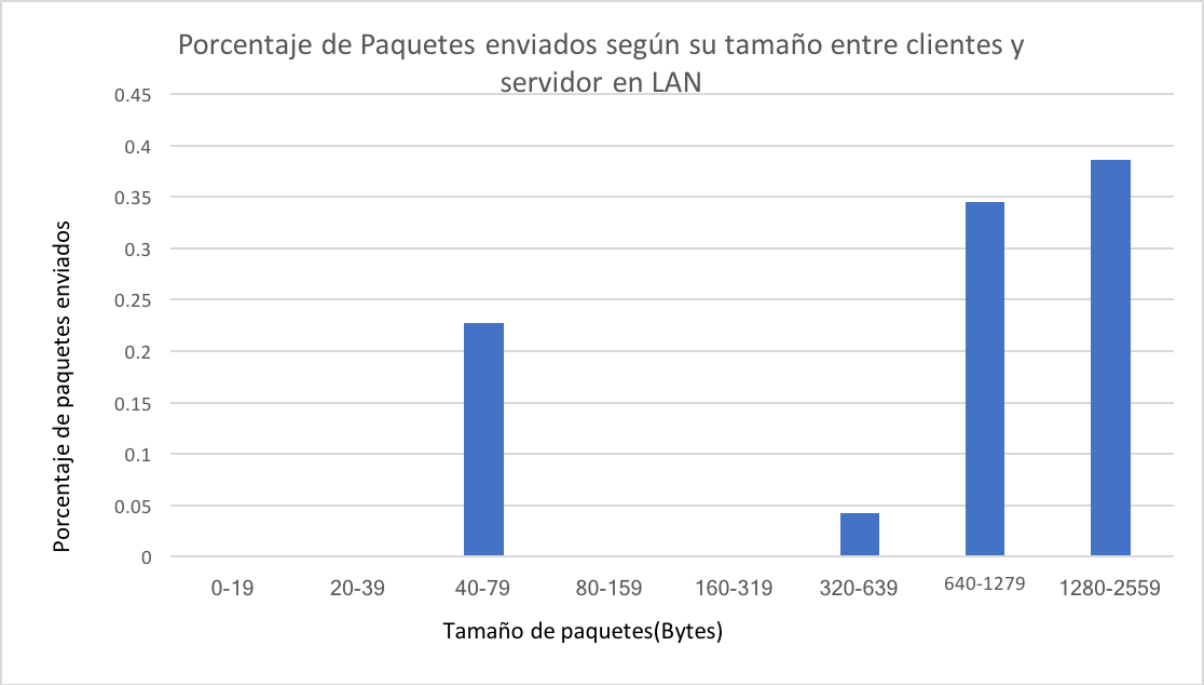

Figura 16: Porcentaje de paquetes enviados según su tamaño entre clientes y servidor.

Conclusiones del análisis

Como conclusiones de estos experimentos se puede decir que el sistema tanto el servidor como el cliente siguen una ecuación casi constante para todos los casos, con una estabilidad luego del intervalo número 200.

En el caso de la memoria usada por el proceso de servidor se ha encontrado una diferencia pronunciada entre máximos y mínimos locales de hasta 1.12 Mb, pero de 1.5 MB para el servidor de multijugador en LAN. Este comportamiento de generar picos de memoria constantemente de una manera periódica se debe al uso del Recolector de Basura de Java que al usar estrategias de eliminación normal y eliminación con compresión permite que los objetos que no se han usado después de cierto tiempo se eliminen para dar prioridad a los más jóvenes (Oracle, n.d.). Este comportamiento del recolector de basura puede ser justificado porque el servidor refresca los objetos de juego que mandan los clientes por actualizaciones del manejador de eventos, lo que significa que el mapa que se encarga de mantener los objetos de juego va a estar cambiando referencias de objetos por nuevos actualizados cada que un cliente haga una conexión, además que el proceso de intercambio, lectura y escritura, requiere la creación de un *SocketChannel* y de un objeto de flujo de entrada o salida de datos que sólo van a ser usados una sola vez cuando los clientes requieran.

En el análisis de la memoria para los clientes se mostró una tendencia creciente, al igual que el servidor, pero con diferencia entre picos de 0.65 Mb lo que muestra un comportamiento de liberación de memoria menos agresivo que en el servidor, además con un mínimo de 9 Mb usados para toda la vida del proceso y un máximo de 26 Mb. La explicación para este comportamiento periódico es debido al recolector de basura de Java que es llamado cada que la pantalla se refresca, pues Processing , en su PApplet, va a

dibujar y a usar nuevos objetos, en este caso cuadrados, para poblar de nuevo al lienzo que se refresca en el lazo principal, dado por la función `draw`(Reas, 2007). Sin embargo, en el experimento de multijugadores en LAN, la memoria de los clientes ha presentado una tendencia a disminuir el uso de más memoria, esto puede ser el resultado de que el recolector de basura usa estrategias de comprensión en vez de eliminación directa para mantener los objetos que se actualizan en los clientes, pues sólo es uno el que representa al jugador en el mapa de juego.

Las conclusiones obtenidas del análisis del uso del CPU muestran que ambos procesos, el cliente y el servidor, tienden a tener un uso casi constante del CPU. Para el caso del servidor para todas las pruebas, excepto la de múltiples clientes en LAN, se obtuvo una tendencia de uso del CPU de 4%, y en los clientes un uso del procesador aproximada al 13%. En el caso del uso del CPU en LAN con múltiples clientes el uso subió a 6.4% que es mayor que las anteriores pruebas, lo cual es esperado porque el servidor tendría que ocuparse de más peticiones por el número de clientes y servir a cada uno según su orden en el selector. Sin embargo, algo que comparten todos los experimentos es un incremento del uso de CPU en los primeros 200 intervalos que es donde se crean y se instancian la mayoría de objetos que los procesos han de usar, como en el cliente que se inician los objetos de juego y propiedades de Processing en el método `setup`.

CONCLUSIONES

Si bien el presente trabajo ha descrito el funcionamiento y diseño de varios sistemas para la organización y combinación de estos en un motor de juegos, también ha mostrado el comportamiento de recursos que el motor ha de requerir en una computadora. Las integraciones de todos los sistemas son importantes por el hecho de que un juego requiere interacción entre el jugador y el mundo del juego, y estos sistemas ayudan a que esto suceda.

Para que un jugador pueda mover a su personaje a través de la pantalla necesita comunicarse al juego por medio del teclado. Para que el sistema entienda que el usuario quiere mover al objeto, este recibe eventos que se transmiten al servidor, donde el sistema guarda la lógica y que a su vez dicte lo que va a pasar a continuación con el objeto si interactúa con otros en el mundo del juego. Para que los objetos interactúen o actúen de cierta manera en el mundo han de tener componentes que son dados por el modelo de objetos. Sin embargo, como la arquitectura es del tipo cliente-servidor, la parte de la lógica y actualización de componentes se encuentra en el servidor, por lo que los eventos generados por el usuario han de ser enviado por red hacia el servidor. Este envío es posible por el uso del sistema de red para múltiples jugadores que usa canales para representar conexiones y selectores para resolver el tipo de operaciones que se necesitan (escritura o lectura).

Sin embargo, el sistema también ha sido probado para definir los requerimientos de red, de procesamiento y de memoria lo que ha mostrado un comportamiento peculiar. El tamaño y número de paquetes enviados muestran un comportamiento optimo pues la mayoría de paquetes son en promedio de 2 KB y 270 paquetes enviados en un segundo por el sistema

lo que no congestiona la red ni causa problemas de latencia en el sistema. Sin embargo, el uso de la memoria ha producido un comportamiento de uso y liberación periódico que no se ha podido atribuir a las causas comunes de liberación de memoria por recolectores de basura de Java. Así mismo, para procesos separados se mostró que los clientes siempre tienen un uso de procesador más alto que el del servidor, 13% y 4% respectivamente. Lo que hace suponer que la ejecución del cliente consume más recursos que la del sistema en general por el uso de Processing en el cliente.

Por otra parte, el sistema aquí descrito puede mejorarse para poder competir con otros motores de juegos comerciales. Por haberse concebido como sistema extensible, el motor de juegos puede mejorar sus capacidades de programación de escenarios, personajes y efectos al aumentar componentes al modelo de objetos. Además, para mejorar la experiencia de usuarios conectados se podría depurar el sistema para servidores lejanos. Sin embargo, dichos cambios podrían requerir más recursos humanos y técnicos.

Por último, el desarrollo de este trabajo espera atraer la atención de nuevos ingenieros al campo del desarrollo de videojuegos. De acuerdo con la página de Gamedevmap, una página de en el Ecuador sólo existen dos empresas de videojuegos. Es por esto que los conocimientos aquí expuestos puedan ser expuestos más abiertamente en el país y así mejorar la riqueza de profesionales en el área.

REFERENCIAS BIBLIOGRÁFICAS

Reas, C. (2007). *Processing A programming Handbook for Visual Designers and Artists.* Cambridge: MIT Press.

DeLoura, M. A. (2001). *Game Programming Gems 2* (Vol. 2). Hingham, Massachusetts, USA: Charles River Medica INC.

Hasiotis, N. (2015, Noviembre 27). *Java Nio Socket Example*. Retrieved Abril 6, 2017, from Java Code Geeks: https://examples.javacodegeeks.com/core-java/nio/java-nio-socket-example/

Cairó, O. (2006). *Estructura de Datos* (3 ed.). México: McGrawHill.

Gregory, J. (2014). *Game Engine Architecture* (2nd ed.). CRC.

PaulDeitel. (2015). *Java How to program* (10 ed.). Pearson.

Silberschtz, G. G. (2012). *Operating System Concepts* (8 ed.). USA: John Wiley & Songs INC.

Sommerville, I. (2000). *Software Engineering* (6th ed.). England: Adison Wesley.

Tanenbaum. (2011). *Computer Networks* (5 ed.). USA: Prentice Hall.

Oracle. (2016). *Selector(Java Platform SE 7)*. Retrieved 02 11, 2017, from Java™ Platform, Standard Edition 7 API Specification: https://docs.oracle.com/javase/7/docs/api/java/nio/channels/Selector.html

Oracle. (n.d.). *Java Garbage Collection Basics*. Retrieved 03 27, 2017, from Oracle: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

Oracle. (2016). *PriorityBlockingQueue*. Retrieved Abril 19, 2017, from Java SE7: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html

Roberts, D. (2015, 09 15). Object Centric Models. *Game Engine Foundations*. Raleigh, NC, USA.

Unity. (2017, Marzo 29). *Unity Manual*. Retrieved Abril 18, 2017, from Unity3D: https://docs.unity3d.com/Manual/index.html?_ga=1.52214889.1336573092.1492525367

R. P., & B. M. (2015). *Software Engineering A Practitioner's approach* (edición 8 ed.). McGrawHill.

Schell. (2008). *The Art of Game Design A Book for Lenses.* Morgan Kaufman.

Bithell. (2012, Junio 30). *mikebithellgames*. Retrieved Abril 18, 2017, from Thomas was alone: http://www.mikebithellgames.com/thomaswasalone/

Weiss, M. A. (2012). *Data Structures and Algorithm Analysis in Java.* Pearson.

Giancoli. (2008). *Física para ciencias e ingeniería.* Pearson.

Processing. (n.d.). Retrieved Abril 20, 2017, from Processing: https://processing.org

Gamedevmap. (2017, Abril 24). Retrieved Mayo 1, 2017, from Gamedevmap: https://www.gamedevmap.com/index.php?country=Ecuador&state=&city=&query=&type=